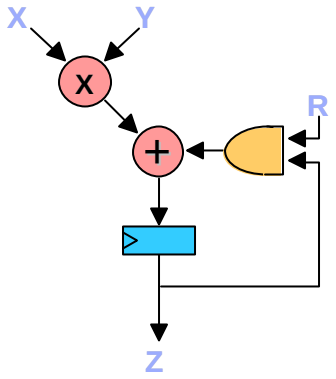
MAC0 - simple, binary




```
input signed [1] R;
input  [w] X,Y;
wire   [accw] XY = X*Y;
wire   [accw] ACCin = XY + (Z&R);
output  [accw] Z = sreg(ACCin);
```

With Module Compiler there is always a **one-to-one correspondence** between the input MCL code and the architecture of the synthesized circuit.

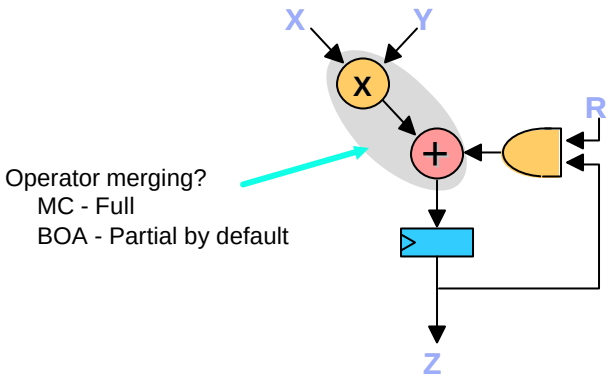
Each operator maps directly to a built-in gate level micro architecture.
This is **architectural synthesis**.

This is a simple Multiply-Accumulator or MAC. The “AND” gate is to clear the accumulator with a reset signal.

In MAC0, the critical path is clearly from the X and Y inputs through the multiplier and adder. It has two carry propagate structures, one carry propagate adder from the multiplier and the other external carry propagate adder for the accumulation.
Therefore, there are totally two carry propagations in series on the critical path.

MAC0 is what you would get from DC.

MAC1 - simple, binary




```
input signed [1] R;
input  [w] X,Y;
wire   [accw] ACCin = X*Y + (Z&R);
output [accw] Z = sreg(ACCin);
```

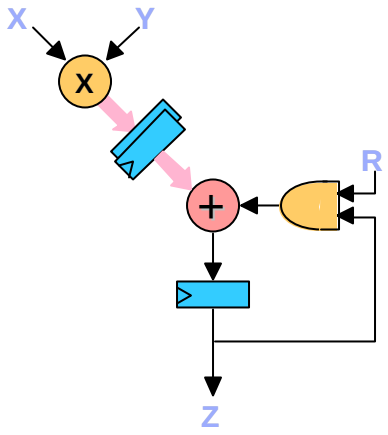
In MAC1, by writing the multiply-add expressions on one line we are exploiting MC's operator merging feature.

The bolded regions are where operator merging has occurred. This eliminates one carry propagate adder from the critical path. Therefore, now there is just one carry propagate structure in the critical path resulting in the speed improvement.

In this case, operator merging is preventing us from pipelining the design. We cannot pipeline the design as the adder (part of the merged operation) is in the feedback loop and therefore pipelining it, would change the functionality.

However, the critical path still is the same, it is clearly from the X and Y inputs through the merged multiplier and propagate adder. What can we do to speed this up?

MAC2 - carrysave multiplier




```
directive local (carrysave="convert");
wire   [accw] prod  = X*Y;
wire   [accw] ACCin = sreg(prod) + (Z&R);
output [accw] Z = sreg(ACCin);
```

The first thing we can do is pipeline the multiplier.

In MAC2, by using the carrysave directive we can leave the result of multiplication in the redundant binary/carrysave format.

Here, we set the carrysave directive to “on” and it will automatically expand the number registers to hold both the carry and sum bits so that we don’t have to put the propagate adder back into the multiplier. This can result in more then double the register count.

In this case too like the previous one, we have only one carry propagate structure in the critical path, however now we can perform pipelining unlike the previous case. Here, the multiply operation (*) can be pipelined since it’s outside the feedback loop. Now, the critical path is most likely the loop which includes the propagate adder and there is no way to pipeline it in this architecture.

Accessing Carrysave Signals Individually ?

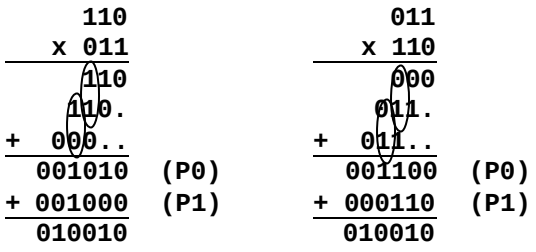
- This is for experienced designers only
- csconvert() is undocumented because it is easy to get into trouble.
- It is only needed when you need to gate or mux signals in carrysave format

The csconvert() function is used as follows

```
wire unsigned [3] X,Y;
wire unsigned [6] Z;
directive(carrysave=convert,multtype=nonbooth);
wire [1] prod = X*Y; // width&format irrelevant
wire unsigned [6] P0,P1; // width & format relevant
csconvert(P0,P1,prod);
directive(carrysave=off);
Z = P0+P1; // binary result
```

Modeling Restrictions of CS Terms P0 and P1

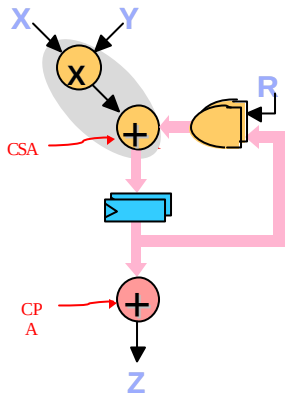
- **The individual bits of P0 and P1 can not be modeled at the RT Level of abstraction. They can only be modeled structurally.**
- **The arithmetic SUM of P0 and P1 can be modeled at the RT Level of abstraction.**
- **Consider 6*3 and 3*6**



In this example of 6 *3 versus 3 * 6, note that P0 and P1 are not just simply reversed, but the are completely different values.

The ellipses represents half-adder cells to do the 3-to-2 reduction in to carry and save terms.

MAC3 - carrysave accumulator



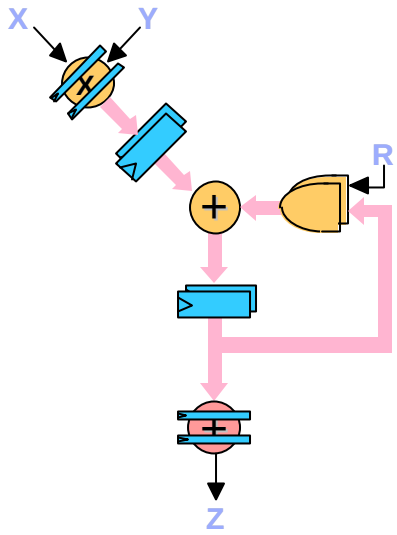

```
wire   [accw] ACC0,ACC1,ACCin0,ACCin1;
directive local (carrysave="convert");
wire   [accw] ACCin = X*Y + (ACC0&R) + (ACC1&R);
csconvert(ACCin0,ACCin1,ACCin);
ACC0 = sreg(ACCin0);
ACC1 = sreg(ACCin1);
output [accw] Z = ACC0+ACC1;
```

Here, we set the carrysave attribute to “convert” which lets us individually access the carry and the sum terms. By doing so, both the multiplier as well as the final propagate adder are out of the feedback loop.

If this is not fast enough, we can easily pipeline the multiply and the propagate adder since they are both out of the feedback loop.

Note that there is a maccs() function in MC that will allow you to do this simple mac with out needing to access the carry and save terms directly. This is the preferred approach.

MAC4 - carrysave multiplier and accumulator




```
directive(fatype=fa, pipeline="on");
wire   [accw] ACC0,ACC1,ACCin0,ACCin1;
wire   [accw] prod,prodR;
directive local (carrysave="convert");
prod  = X*Y;
prodR = ResolveLatencyLoop(prod,3);
directive local(carrysave="convert",pipeline="off");
wire   [accw] ACCin = prodR + (ACC0&R) + (ACC1&R);
csconvert(ACCin0,ACCin1,ACCin);
ACC0 = sreg(ACCin0);
ACC1 = sreg(ACCin1);
directive local (pipeline="on");
wire [accw] Z_tmp = ACC0+ACC1;
output [accw] Z = ResolveLatency(Z_tmp,2);
```

For this last example, pipelining is turned on and delay set such that we get two pipe stages on both the multiplier and the propagate adder. This would run faster with more pipeline stages until it hit the limit of the loop delay. This is the fastest MAC architecture.

In this case we have used the ResolveLatencyLoop() to set two pipe stages in the multiplier. ResolveLatencyLoop(), unlike the ResolveLatency() registers the output. This ensures that while auto-pipelining is on, no portion of the multiplier logic gets into the loop which is now the critical path. Hence, the critcal path is strictly restricted to the elements in the loop. The loop cannot be pipelined, however the final adder outside the loop can be pipelined. Finally we make use of the ResolveLatency() to set two pipe stages for this adder.